Computer Science Department

College of Computer at Al-Lith

جــامـعـة أم الــقـرى

UMM AL-QURA UNIVERSITY

Data Structures

Chapter 2: Linear Data Structure

Instructor Maher Hadiji

maher.hdiji@gmail.com

2015-2016

Outline

1 Classification of Data Structure

2 Linear Data Structures

1 Array

2 Linear List

1.Singly-linked lists

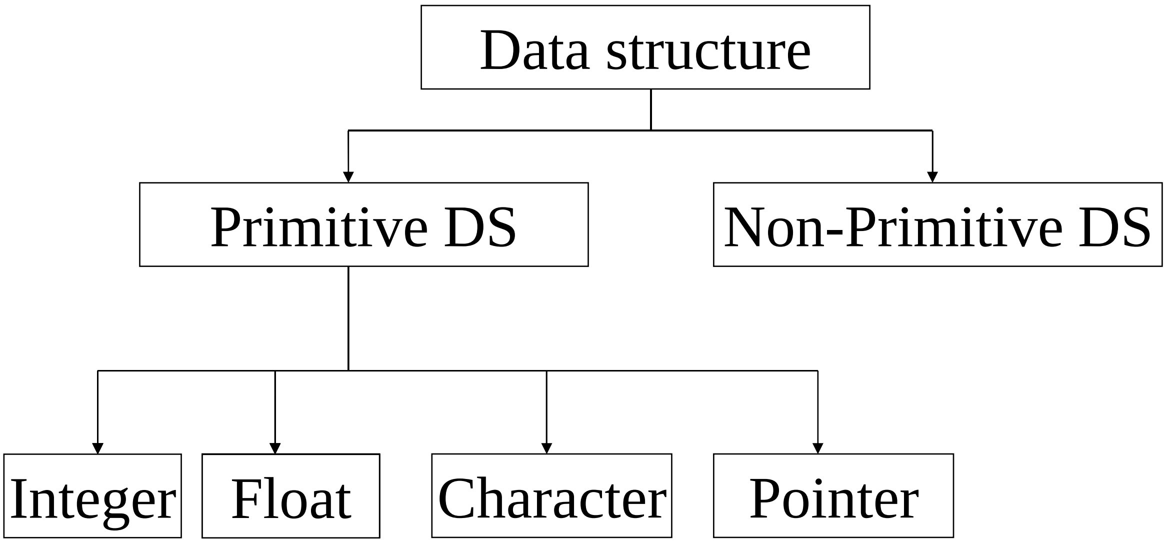
2.Doubly-linked lists

3 Queue / stack

1.Queue

2. stack

1 Classification of Data Structure



Data structure

Primitive DS

Non-Primitive DS

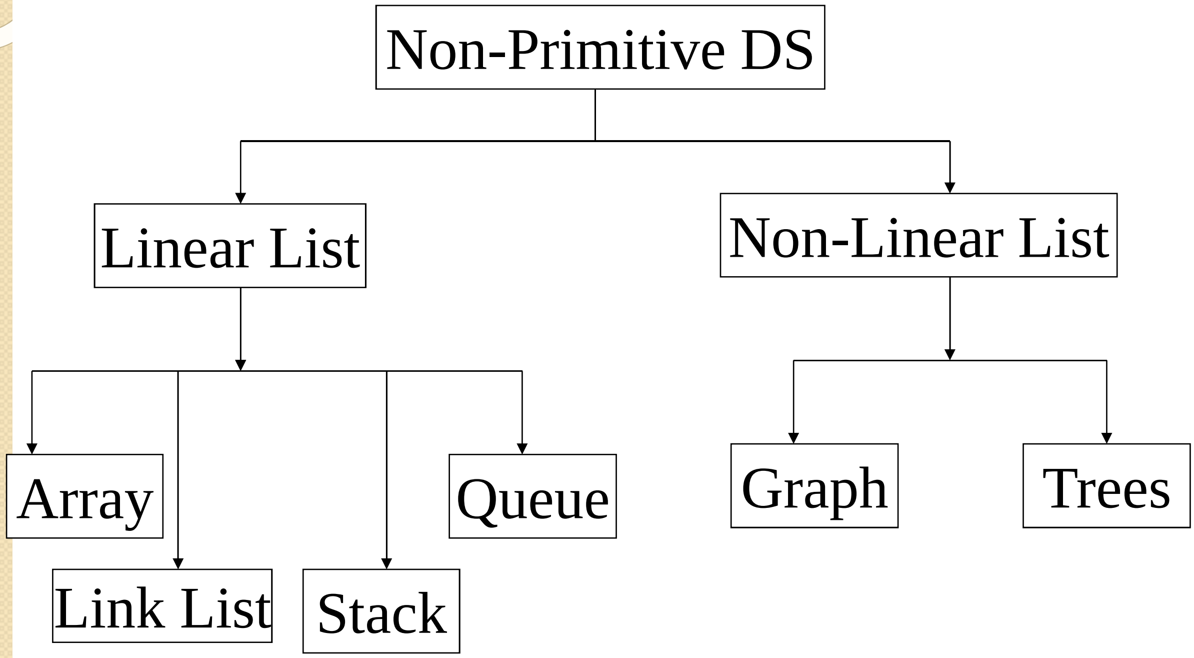
Integer

Float

Character

Pointer

1 Classification of Data Structure



Non-Primitive DS

Linear List

Non-Linear List

Array

Link List

Stack

Queue

Graph

Trees

Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- In general, there are different representation on different computers.
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

Non-Primitive Data Structure

- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.

Non-Primitive Data Structure

- Lists, Stack, Queue, Tree, Graph are example of non-primitive data structures.
- The design of an efficient data structure must take operations to be performed on the data structure.

Non-Primitive Data Structure

- The most commonly used operation on data structure are broadly categorized into following types:
  - Create
  - Selection
  - Updating
  - Searching
  - Sorting
  - Merging
  - Destroy or Delete

Different between them

- A primitive data structure is generally a basic structure that is usually built into the language, such as an integer, a float.
- A non-primitive data structure is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree, AVL Tree, graph etc.

2 Linear Data Structures

- Arrays
  - A sequence of n items of the same data type that are stored **contiguously** in computer memory and made accessible by specifying a value of the array's **index**.
- Linked List
  - A sequence of zero or more nodes each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list.
  - Singly linked list (next pointer)
  - Doubly linked list (next + previous pointers)

- Arrays
  - fixed length (need preliminary reservation of memory)
  - contiguous memory locations
  - direct access
  - Insert/delete
- Linked Lists
  - dynamic length
  - arbitrary memory locations
  - access by following links
  - Insert/delete

2.1 Array

- The Array is the most commonly used Data Storage Structure.
- It’s built into most Programming languages.

Creating an Array

- An array is a sequential data abstrction, its name is a reference to an array.

```
int[ ] intArray; //defines a reference to an
  array
intArray = new int[100];  //creates the array
```

INITIALIZATION

- In Java, an array of integers is automatically initialized to 0.
- Unless you specify otherwise,
- You can initialize an array to something beside 0 using this syntax:

```
int[] intArray ={0,1,2,3,4,5,6,7,8,9};
```

Accessing Array Elements

- Array elements are accessed using an index number.

```
temp = intArray[3]; //get 4th element content
intArray[7] = 66; //insert 66 in eighth cell
```

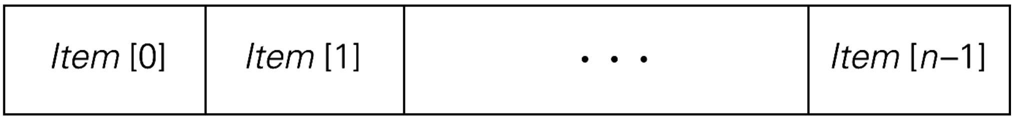
Example

```
class ArrayApp
   { public static void main(String[] args)
    {   long[] arr;                  // reference to array
      arr = new long[100];         // make array
      int nElems = 0;              // number of items
      int j;                       // loop counter
    long searchKey;   // key of item to search for
//-----------------------------------------------
      arr[0] = 77;          // insert 10 items
      arr[1] = 99;
      arr[2] = 44;
      arr[3] = 55;
     arr[4] = 22;
      arr[5] = 88;
      arr[6] = 11;
      arr[7] = 00;
      arr[8] = 66;
      arr[9] = 33;
      nElems = 10;
  // now 10 items in array
```

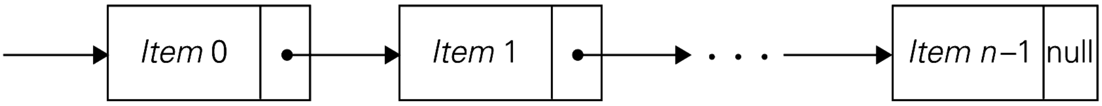
```
  //------------------------------------------------------------
        for(j=0; j<nElems; j++)       // display items
          System.out.print(arr[j] + " ");
        System.out.println("");
  //------------------------------------------------------------
        searchKey = 66;              // find item with key 66
        for(j=0; j<nElems; j++)          // for each element,
          if(arr[j] == searchKey)        // found item?
            break;                     // yes, exit before end
   if(j == nElems)                    // at the end?
          System.out.println("Can't find " + searchKey); // yes
        else
          System.out.println("Found " + searchKey);      // no
```

```
//-----------------------------------------------
    searchKey = 55;              // delete item with key 55
    for(j=0; j<nElems; j++)           // look for it
    if(arr[j] == searchKey)
      break;
    for(int k=j; k<nElems; k++)       // move higher ones down
      arr[k] = arr[k+1];
   nElems--;                         // decrement size
//-----------------------------------------------------------
   for(j=0; j<nElems; j++)       // display items
```

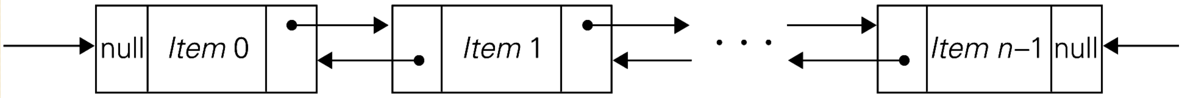
- `System.out.print( arr[j] + " ");`
- `System.out.println("");`
- `} // end main()`
- `} // end class ArrayApp`



Array of *n* elements



Singly linked list of *n* elements

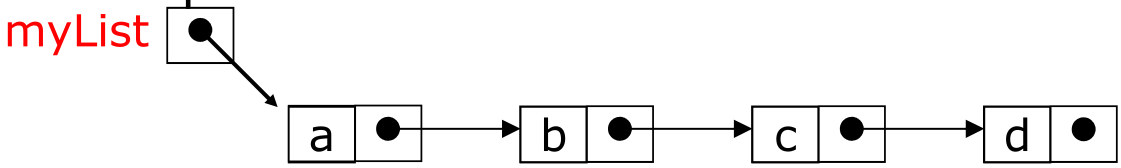


Double linked list of *n* elements

2.2 Linked Lists

Anatomy of a linked list

- A linked list consists of:
  - A sequence of nodes



Each node contains a value
and a link (pointer or reference) to some other node

The last node contains a null link

The list may have a header

More terminology

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)

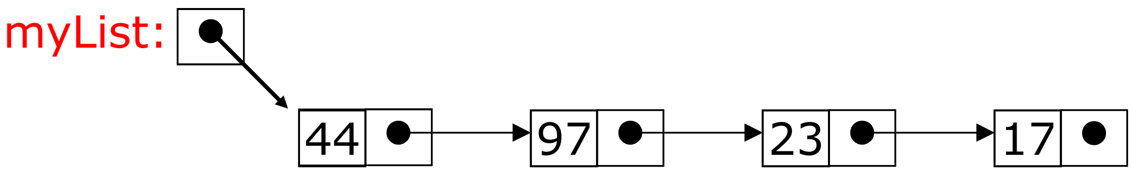
Pointers and references

- In Java, a reference is more of a “black box,” or ADT
  - Available operations are:
    - dereference (“follow”)
    - copy
    - compare for equality
  - There are constraints on what kind of thing is referenced: for example, a reference to an array of int can *only* refer to an array of int

Creating references

- The keyword new creates a new object, but also returns a *reference* to that object
- For example, Person p = new Person("John")
  - new Person("John") creates the object and returns a reference to it
  - We can assign this reference to p, or use it in other ways

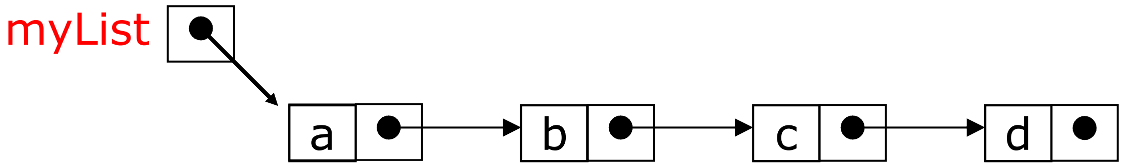
Creating links in Java



```
class Cell { int value;
             Cell next;
   Cell (int v, Cell n) { // constructor
       value = v;
       next = n;
   }
}
Cell temp = new Cell(17, null);
temp = new Cell(23, temp);
temp = new Cell(97, temp);
Cell myList = new Cell(44, temp);
```

2.2.1 Singly-linked lists

- Here is a singly-linked list (SLL):



- Each node contains a value and a link to its successor (the last node has no successor)
- The header points to the first node in the list (or contains the null link if the list is empty)

Singly-linked lists in Java

```
public class SLL {

   private SLLNode first;

   public SLL() {
        this.first = null;
   }

   // methods...
}
```

- This class actually describes the *header* of a singly-linked list
- However, the entire list is accessible from this header
- Users can think of the SLL as *being* the list
  - Users shouldn’t have to worry about the actual implementation

SLL nodes in Java

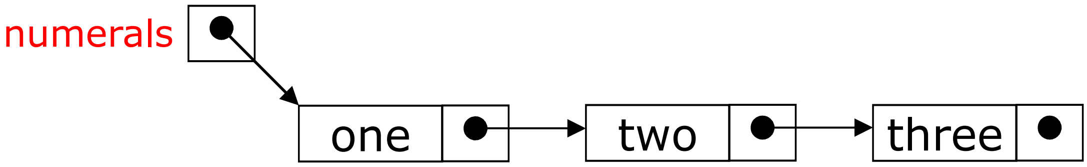
```
public class SLLNode {
   protected Object element;
   protected SLLNode succ;

   protected SLLNode(Object elem, SLLNode succ) {
     this.element = elem;
     this.succ = succ;
    }
}
```

Creating a simple list

- To create the list ("one", "two", "three"):
- SLL numerals = new SLL();
- numerals.first =

```
 new SLLNode("one",
        new SLLNode("two",
            new SLLNode("three",
null)));
```



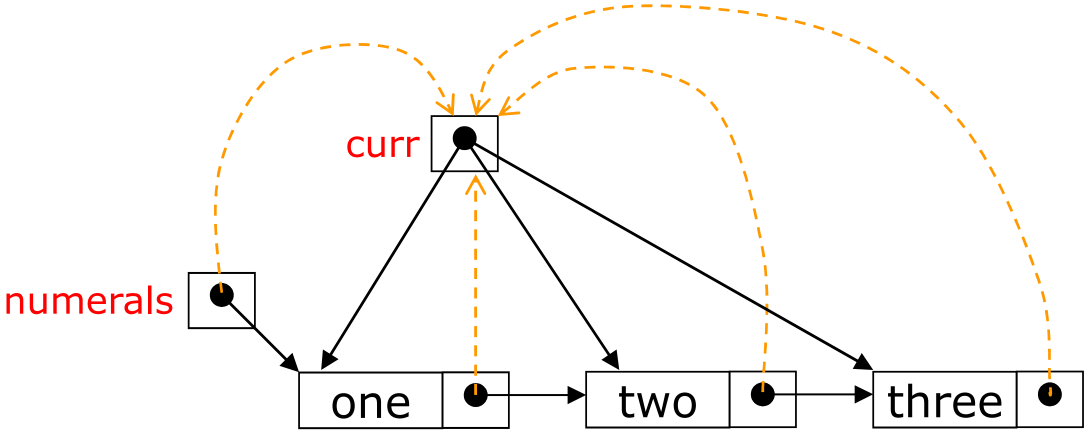
Traversing a SLL

- The following method traverses a list (and prints its elements):

```
public void printFirstToLast() {
  for (SLLNode curr = first;
        curr != null;
        curr = curr.succ) {
     System.out.print(curr.element + "  ");
   }
}
```

- You would write this as an instance method of the SLL class

Traversing a SLL (animation)



Inserting a node into a SLL

- There are many ways you might want to insert a new node into a list:
  - As the new first element
  - As the new last element
  - Before a given node (specified by a *reference*)
  - After a given node
  - Before a given value
  - After a given value
- All are possible, but differ in difficulty

Inserting as a new first element

- This is probably the easiest method to implement
- In class SLL (not SLLNode):

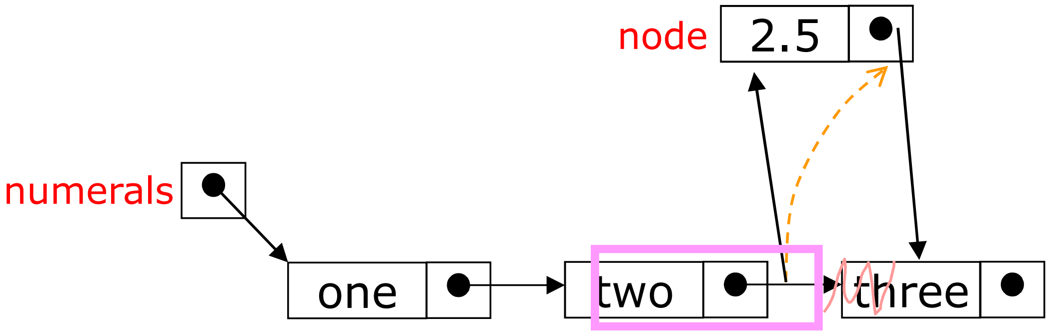
```
void insertAtFront(SLLNode node) {
    node.succ = this.first;
    this.first = node;
}
```

- Notice that this method works correctly when inserting into a previously empty list

Inserting a node after a given value

```
void insertAfter(Object obj, SLLNode node) {
   for (SLLNode here = this.first;
         here != null;
         here = here.succ) {
       if (here.element.equals(obj)) {
          node.succ = here.succ;
          here.succ = node;
          return;
       } // if
   } // for
   // Couldn't insert--do something reasonable!
}
```

Inserting after (animation)



Find the node you want to insert after

First, copy the link from the node that's already in the list

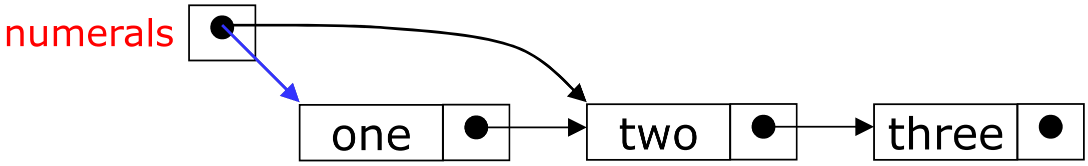
Then, change the link in the node that's already in the list

Deleting a node from a SLL

- In order to delete a node from a SLL, you have to change the link in its *predecessor*
- This is slightly tricky, because you can't follow a pointer backwards
- Deleting the first node in a list is a special case, because the node's predecessor is the list header

Deleting an element from a SLL

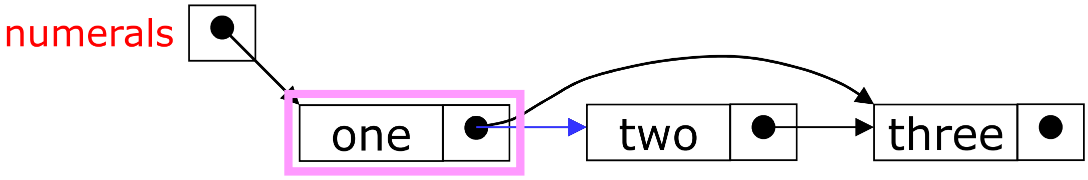
- To delete the first element, change the link in the header



numerals

one two three

- To delete some other element, change the link in its predecesso



numerals

one two three

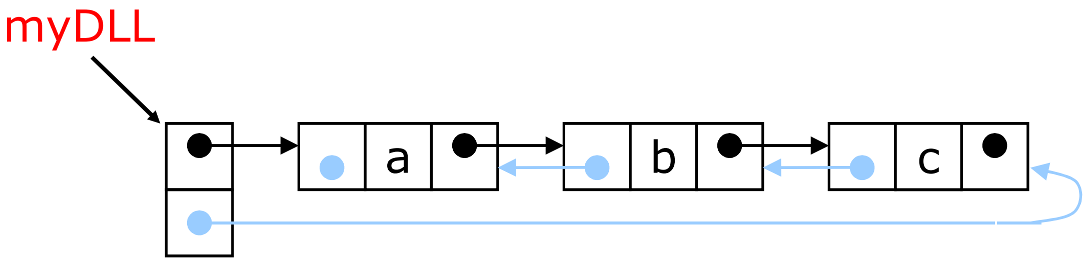
- Deleted nodes will eventually be garbage collected

Deleting from a SLL

```
public void delete(SLLNode del) {
  SLLNode succ = del.succ;
  // If del is first node, change link in header
  if (del == first) first = succ;
  else { // find predecessor and change its link
    SLLNode pred = first;
    while (pred.succ != del) pred = pred.succ;
    pred.succ = succ;
  }
}
```

2.2.2Doubly-linked lists

- Here is a doubly-linked list (DLL):



- Each node contains a value, a link to its successor (if any), and a link to its predecessor (if any)
- The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)

DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Constructing SLLs and DLLs

```
public class SLL {

   private SLLNode
   first;

   public SLL() {
         this.first = null;
   }
   // methods...
}
```

```
public class DLL {
   private DLLNode
   first;
   private DLLNode
   last;
   public DLL() {
         this.first = null;
         this.last = null;
   }
   // methods...
}
```

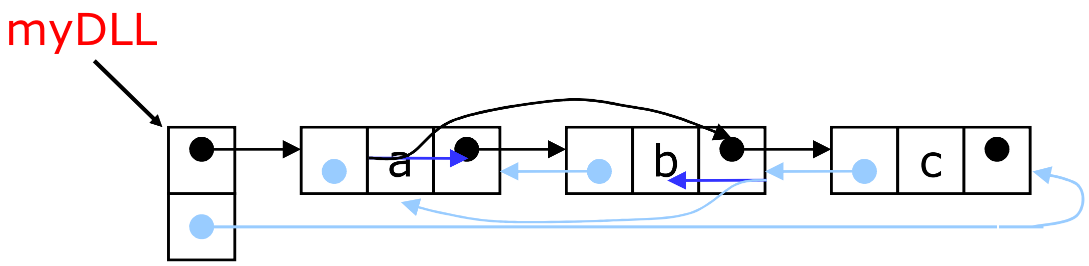
DLL nodes in Java

```
public class DLLNode {
  protected Object element;
  protected DLLNode pred, succ;

  protected DLLNode(Object elem,
                    DLLNode pred,
                    DLLNode succ) {
    this.element = elem;
    this.pred = pred;
    this.succ = succ;
   }
}
```

Deleting a node from a DLL

- Node deletion from a DLL involves changing *two* links



- Deletion of the first node or the last node is a special case
- Garbage collection will take care of deleted nodes

Other operations on linked lists

- Most "algorithms" on linked lists—such as insertion, deletion, and searching—are pretty obvious; you just need to be careful
- Sorting a linked list is just messy, since you can't directly access the nth element—you have to count your way through a lot of other elements

2.3 Stacks, Queues (1)

Application: recursive function to save parameters

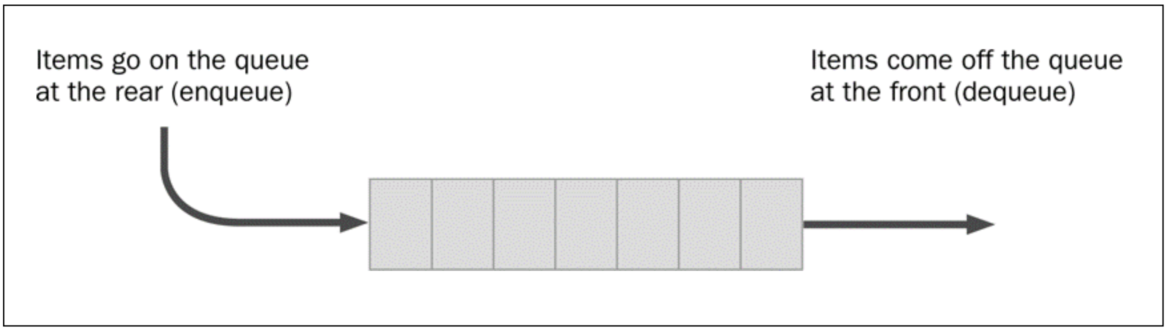
- Stacks
  - A stack of plates
    - insertion/deletion can be done only at the top.
    - LIFO/FILO
  - Two operations (push and pop)
- Queues
  - A queue of customers waiting for services
    - Insertion/enqueue from the rear and deletion/dequeue from the front.
    - FIFO
  - Two operations (enqueue and dequeue)

2.3.Stacks, Queues

- Priority queues (implemented using heaps)
  - A data structure for maintaining a set of elements, each associated with a key/priority, with the following operations
    - Finding the element with the highest priority
    - Deleting the element with the highest priority
    - Inserting a new element
  - Scheduling jobs on a shared computer.

2.3.1 Queues

- A *queue* is a list that adds items only to the rear of the list and removes them only from the front
- It is a FIFO data structure: First-In, First-Out
- Analogy: a line of people at a bank teller's window


Items go on the queue at the rear (enqueue)
Items come off the queue at the front (dequeue)

2.3.1 Queues

- Classic operations for a queue
  - enqueue - add an item to the rear of the queue
  - dequeue (or serve) - remove an item from the front of the queue
  - empty - returns true if the queue is empty
- Queues often are helpful in simulations or any situation in which items get "backed up" while awaiting processing

2.3.1Queues

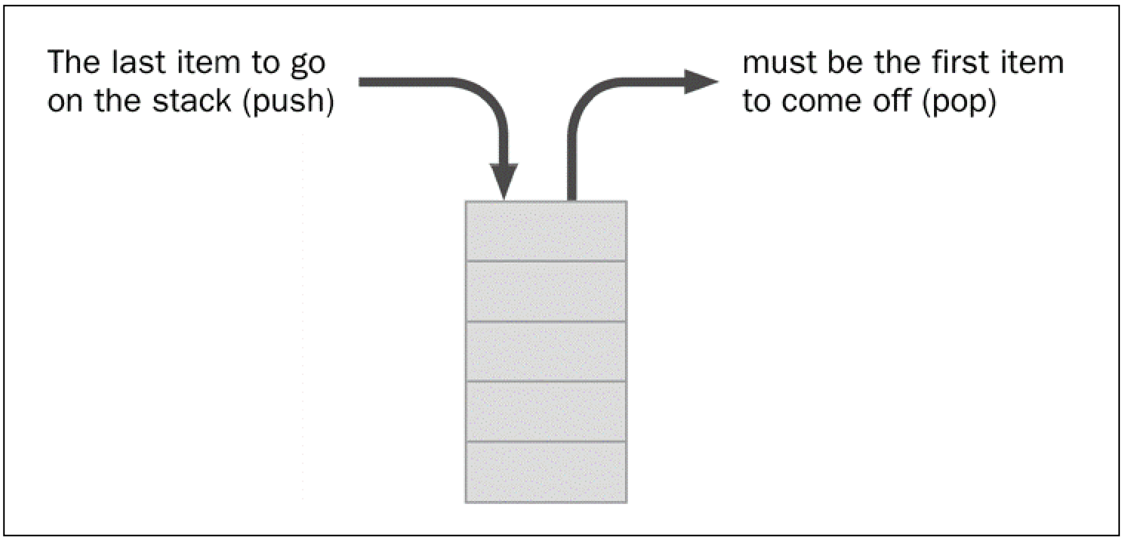
- A queue can be represented by a singly-linked list; it is most efficient if the references point from the front toward the rear of the queue
- A queue can be represented by an array, using the remainder operator (%) to "wrap around" when the end of the array is reached and space is available at the front of the array

2.3.2Stacks

- A *stack* ADT is also linear, like a list or a queue
- Items are added and removed from only one end of a stack
- It is therefore LIFO: Last-In, First-Out
- Analogies: a stack of plates or a stack of books

2.3.2Stacks

- Stacks often are drawn vertically:



The last item to go on the stack (push)

must be the first item to come off (pop)

2.3.2 Stacks

- Clasic stack operations:
  - push - add an item to the top of the stack
  - pop - remove an item from the top of the stack
  - peek (or top) - retrieves the top item without removing it
  - empty - returns true if the stack is empty
- A stack can be represented by a singly-linked list, with the firs node in the list being to top element on the stack
- A stack can also be represented by an array, with the bottom the stack at index 0

2.3.2Stacks

- The `java.util` package contains a `Stack` class
- The `Stack` operations operate on `Object` references
- Suppose a message has been encoded by reversing the letters of each word
- See `Decode.java`

```
//********************************************************************
//  Decode.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the Stack class.
//********************************************************************

import java.util.*;

public class Decode
{
   //-----------------------------------------------------------------
   //   Decodes a message by reversing each word in a string.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Scanner scan = new Scanner (System.in);

      Stack word = new Stack();

      String message;
      int index = 0;

      System.out.println ("Enter the coded message:");
      message = scan.nextLine();
      System.out.println ("The decoded message is:");
```

continue

continue

```
      while (index < message.length())
      {
         // Push word onto stack
         while (index < message.length() && message.charAt(index) != '
')
         {
            word.push (new Character(message.charAt(index)));
            index++;
         }

         // Print word in reverse
         while (!word.empty())
            System.out.print (((Character)word.pop()).charValue());
         System.out.print (" ");
         index++;
      }

      System.out.println();
   }
}
```

Sample Run

```
Enter the coded message:
artxE eseehc esaelp
The decoded message is:
Extra cheese please
```

continue

```
      while (index < message.length())
      {
         // Push word onto stack
         while (index < message.length() && message.charAt(index) != ' ')
         {
            word.push (new Character(message.charAt(index)));
            index++;
         }

         // Print word in reverse
         while (!word.empty())
            System.out.print (((Character)word.pop()).charValue());
         System.out.print (" ");
         index++;
      }

      System.out.println();
   }
}
```